

# Dynamic Deadlock Detection under the OR Requirement Model

Alvaro E. Campos, Christian F. Orellana\*, and María Pía Soto

Departamento de Ciencia de la Computación

Pontificia Universidad Católica de Chile

Casilla 306 - Santiago 22 - CHILE

## ABSTRACT

Deadlock detection is one of the most discussed problems in the literature. Although several algorithms have been proposed, the problem is still open. In general, the correct operation of an algorithm depends on the requirement model being considered. This article introduces a deadlock-detection algorithm for the OR model. The algorithm is complete, because it detects all deadlocks, and it is correct, because it does not detect false deadlocks. In addition, the algorithm supports dynamic changes in the wait-for graph on which it works. Once finalized the algorithm, at least each process that causes deadlock knows that it is deadlocked. Using this property, possible extensions are suggested in order to resolve deadlocks.

**Keywords:** Distributed systems, deadlock detection, deadlock resolution, self-stabilization, wait-for graph, knot.

## 1. INTRODUCTION

One of the main motivations to build distributed systems is the possibility of sharing resources among several processors. A process can acquire and release resources in a sequence that is unknown beforehand. In this setting, the deadlock problem arises; being able to detect deadlocks is the first step to take actions and resolve them.

A set of processes is said to be deadlocked when each process in the set is blocked, waiting for resources that are assigned to other processes in the same set [14]. This situation may occur if four conditions related to resource competition hold simultaneously in the system: resources cannot be shared, there is no preemption, processes waiting for resources keep those already assigned to them, and there is circular wait.

In general, there are three strategies to deal with deadlocks: prevention, avoidance, and detection [14]. The first two strategies free the sys-

tem from the possibility of becoming deadlocked, but are inefficient. The first one imposes restrictions in the way a process can execute, to negate one of the four conditions mentioned above. The second one is computationally expensive, because the system must check the safety of the new state every time a resource is about to be assigned. The third one, on the other hand, lets deadlocks occur, but then it detects and resolves them.

## 2. THEORETICAL MODELS

Knapp classified the deadlock-detection problem in six models, according to the type of requirements that a process can make [10]. The six models are the following:

**Single-outstanding-request model:** Under this model, processes request only one resource at a time. It is the simplest requirement model. Flatebo and Datta, among others, have proposed an algorithm to detect deadlocks under this model [4].

**AND model:** Under this model, processes request multiple resources simultaneously. A requirement is satisfied when all the requested resources are assigned. Some algorithms to detect deadlocks [4, 9] and to resolve them [6] under this model have been proposed.

**OR model:** Under this model, processes also request multiple resources simultaneously, but a requirement is satisfied when any of the requested resources is assigned. Chandy, Misra, and Haas [2], and Natarajan [11], among others, have proposed algorithms to detect deadlocks under this model.

**AND/OR model:** This model is a generalization of the previous two. Under this model, processes request any number of resources in an arbitrary combination of AND and OR requirements. Herman and Chandy have proposed a distributed algorithm to detect deadlocks under this model [7].

---

\*Corresponding author

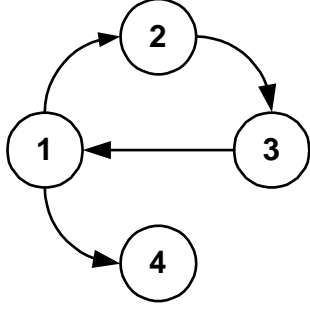


Figure 1: Deadlock under the AND model

**( $\frac{n}{k}$ ) model:** Under this model, a requirement for  $n$  resources is satisfied when  $k$  of them are assigned to the requesting process. It is a generalized model too, because when  $n = k = 1$  the requirement corresponds to the first model described. If  $n = k \neq 1$ , it is an AND requirement, and if  $k = 1 \neq n$ , it is an OR requirement. Bracha and Toueg have proposed an algorithm to detect and resolve deadlocks under this model [1].

**Unrestricted model:** Under this model, no assumption is made about the way in which processes can make their requirements.

This article presents a deadlock-detection algorithm for the OR requirement model. A process can make an OR request, for example, in a replicated distributed database system, where a read request for a replicated element is satisfied when any copy is read [10]. In a similar way, in a message-routing system based on wormhole routing, a router can forward a received message to a neighbor router through one of several channels [13]; a requirement for an output channel is satisfied when any of them becomes available.

A useful way to represent resource requirements is by means of a directed graph, known as *Wait-For Graph* (WFG). In a WFG, each node represents a process in the system. Nodes with outgoing edges represent blocked processes, waiting for resources. On the contrary, nodes without outgoing edges represent active processes. An edge from node  $i$  to node  $j$  means that process  $i$  is waiting for a resource assigned to process  $j$ . The deadlock-detection problem can be reduced to that of detecting cyclic structures on this graph. For example, the presence of a directed cycle in the WFG is a necessary and sufficient condition for the existence of deadlock under the AND model [10]. In Figure 1, processes 1, 2, and 3 form a cycle, and are deadlocked. Although the resource assigned to process 4 will be released once that process terminates, process 1 also needs the resource assigned to process 2, to be able to continue executing.

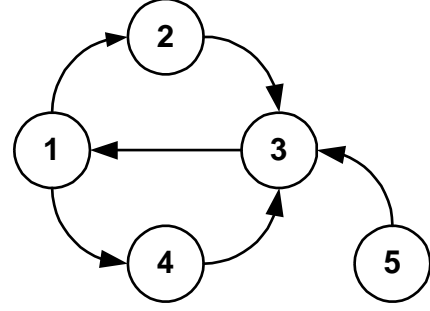


Figure 2: Deadlock under the OR model

Under the OR requirement model, the presence of a cycle in the WFG is a necessary — but not sufficient — condition for a deadlock to exist. If the edges represent OR requirements, there is no deadlock in Figure 1, despite the cycle, because process 1 is waiting for the resource assigned to process 2 or the resource assigned to process 4. Since process 4 can terminate, its resource can be assigned to process 1, which can then continue executing.

Under the OR requirement model, a process is *blocked* if it has a pending OR requirement. The set of processes for which a blocked process  $P$  is waiting is called its *dependent set*; the dependent set contains the direct successors, the *neighbors*, of  $P$  in the WFG [2]. Therefore, a blocked process can continue executing if some process in its dependent set releases a resource, which is then assigned to the blocked process.

Under the OR requirement model, a set  $S$  of processes is in a deadlock state if and only if the following conditions hold [2]:

- every process in  $S$  is blocked,
- the dependent set of each process in  $S$  is a subset of  $S$ , and
- there are no messages in transit between processes in  $S$ .

Detecting a deadlock is equivalent to detecting a *knot* in the WFG [8]. By definition, a node  $v$  is in a knot if,

$$\forall \text{ node } w, v \text{ reaches } w \rightarrow w \text{ reaches } v,$$

that is, if there is a directed path to  $v$  from each one of its successors. Nodes that are in the knot are said to *cause* the deadlock. There may be other nodes, which are not in a knot, that are deadlocked only because all nodes in their dependent sets are deadlocked.

In Figure 2, processes 1, 2, 3, and 4 are in a knot, they are deadlocked, and they all cause deadlock. Although process 5 is not in the knot, it is deadlocked nonetheless, because all processes in its dependent set are.

For node  $i$ :

- (1.1) **if**  $(j, i) \notin Succ \wedge j \in Neighbors$   
**then**  $Succ := Succ \cup \{(j, i)\}$
- (1.2) **if**  $(j, i) \in Succ \wedge j \notin Neighbors$   
**then**  $Succ := Succ - \{(j, i)\}$
- (1.3) **if**  $Neighbors = \emptyset \wedge (Succ \neq \emptyset \vee Paths \neq \emptyset)$   
**then**  $Succ := \emptyset;$   
 $Paths := \emptyset$
- (1.4) **if**  $(j, -) \in Succ_n \wedge (j, -) \notin Succ$   
**then**  $Succ := Succ \cup \{(j, n)\}$
- (1.5) **if**  $(j, n) \in Succ \wedge (j, -) \notin Succ_n$   
**then**  $Succ := Succ - \{(j, n)\}$

Figure 3: The proposed algorithm, first phase

Chandy, Misra, and Haas have proposed an algorithm to detect deadlocks under the OR model, based on the technique known as *diffusing computations* [2]. In their proposal, a process starts the algorithm when a request is not granted. Upon termination, a process is guaranteed to know that it is deadlocked only if it was deadlocked when the algorithm started. Nonetheless, in a set of deadlocked processes, at least one of them is able to report the deadlock. The algorithm proposed by Natarajan [11] is based on the same principle as the one by Chandy, Misra, and Haas, but uses a periodic protocol that allows to choose exactly one process from a set of deadlocked processes to report it.

In the algorithm proposed in this paper, a process that is not part of a knot when the algorithm starts, but later becomes part of one, is able to report the deadlock. If all processes in the dependent set of a process  $v$  are aware that they are deadlocked, process  $v$  is able to report that it is deadlocked. Additionally, each process could decide if it is part of a knot. Thus, all processes that are aware of the deadlock, in particular, those processes that cause the deadlock, may start a resolution action.

### 3. DEADLOCK DETECTION UNDER THE OR MODEL

The proposed algorithm is shown in Figures 3, 4, and 5. Detection is made in three phases. In the first phase, each node computes the set of its successors. In the second phase, each node builds a partial view of the WFG, by the propagation of existing paths. Finally, in the third phase, each node decides locally whether it is deadlocked or not.

The algorithm starts at a node, when a request for resources is not granted. The requesting process blocks, and control is transferred to a thread that runs the detection algorithm. These threads maintain exact, up-to-date information

about which are the processes for whom they wait, that is, their *neighbors* in the WFG. The set of neighbors of a node  $v$  changes when one of them releases a resource, which is then reallocated to some waiting node. If it is reallocated to  $v$ ,  $v$  is no longer blocked; otherwise,  $v$  has a different set of neighbors. The resource allocator can inform the detection-algorithm thread of these changes in the WFG, through atomic updates of the local variable *Neighbors*. No other event can change the set of neighbors, since the process is blocked.

### Variables

Each process maintains four local variables when executing the algorithm: *Neighbors*, *Succ*, *Paths*, and *deadlock*, which it can read and write. In addition, it is assumed that each process has read-only access to the local variables of the processes for whom it waits; actually, read access to variables *Succ*, *Paths*, and *deadlock* of the neighbors is enough.

Variable *Succ* represents the set of successors of the node that is executing the algorithm. Each element of the set is an ordered pair of the form  $(a, b)$ , where  $a$  is the identifier of the successor process, and  $b$  is the identifier of the process from where the information was learned. Variable *Paths* represents the partial view of the WFG that the node has. Each element of this set is an ordered trio of the form  $(a, b, c)$ , and indicates that there is a directed path from node  $a$  to node  $b$ , and that the information was learned from node  $c$ . Variable *deadlock* indicates if the process is deadlocked.

Initially, variable *deadlock* takes the value *false*. Variables *Succ* and *Paths* must be initialized as empty sets.

For node  $i$ :

- (2.1) **if**  $(a, \_) \in Succ \wedge (i, a, i) \notin Paths$   
**then**  $Paths := Paths \cup \{(i, a, i)\}$
- (2.2) **if**  $(a, \_) \notin Succ \wedge (i, a, i) \in Paths$   
**then**  $Paths := Paths - \{(i, a, i)\}$
- (2.3) **if**  $(a, b, \_) \in Paths_n \wedge (a, b, \_) \notin Paths$   
**then**  $Paths := Paths \cup \{(a, b, n)\}$
- (2.4) **if**  $(a, b, n) \in Paths \wedge (a, b, \_) \notin Paths_n$   
**then**  $Paths := Paths - \{(a, b, n)\}$

Figure 4: The proposed algorithm, second phase

## Notation

Each step of the algorithm is of the following form:

$(p.s)$  **if**  $\langle \text{guard} \rangle$  **then**  $\langle \text{move} \rangle$

In the number  $(p.s)$ ,  $p$  indicates the phase to which the step belongs, and  $s$  enumerates the steps within that phase. The predicate  $\langle \text{guard} \rangle$  is a boolean predicate over the variables that the process can read: its own local variables and the ones of its neighbors. If the predicate is true, then it is possible to execute the action defined in  $\langle \text{move} \rangle$ . It is assumed the existence of a coordinator — centralized or distributed — that chooses the next move to execute when there is one or more true predicates. Moves are chosen according to the phase to which they belong; the ones that belong to an earlier phase have priority over actions of later phases.

In Figures 3, 4, and 5, variable  $i$  represents the identifier of the process that is executing the algorithm, and  $n$  represents the identifier of one of its neighbors. The local variables of neighbor  $n$  are represented as  $Succ_n$ ,  $Paths_n$ , and  $deadlock_n$ . Tuples of the form  $(a, \_)$  represent any ordered pair whose first element is  $a$ . In the same way, those of the form  $(a, b, \_)$  represent any ordered trio whose first two elements are  $a$  and  $b$ .

## The algorithm

Phase 1, in Figure 3, begins with step (1.1), which is executed by process  $i$ , when its set of neighbors changes. The set first changes when  $i$  blocks because it makes a request that is not granted and, therefore, it acquires a set of neighbors. The process computes its initial set of successors, with the identifiers of those neighbors. In step (1.2), later changes in the set of neighbors update the set of successors accordingly. In step (1.4), the set of successors propagates to predecessors nodes. The goal of this propagation is to determine all successors of a node, both direct and indirect. The node completes the pairs of its indirect successors with the identifier of the neighbor node from

where the information was learned. In this way, in step (1.5), the node is able to eliminate all tuples that, because of a change in the WFG, are no longer valid. Step (1.3) ends the algorithm, in case one of the requested resource for which the process was waiting for, is assigned. In this case, process  $i$  is no longer blocked and has no neighbors. Upon termination of this phase, each node knows exactly all its successors.

Phase 2, in Figure 4, begins with step (2.1), where the set of paths from node  $i$  to all its successors is completed. In step (2.3), paths are propagated to predecessors nodes in a way similar to the propagation of successors in phase 1. Steps (2.2) and (2.4) propagate changes in the WFG. Upon termination of this phase, each node knows the set of paths in the WFG that start on itself and on every one of its successors, that is, each node has a partial view of the WFG that allows it to decide whether it is or not in a knot.

Phase 3, in Figure 5, begins with step (3.1), which determines the existence of a knot in the WFG. If there is one, the process knows that it is deadlocked. Step (3.2) is for nodes that, without being part of a knot, are waiting only for processes that already know that are deadlocked; they can infer that they are also deadlocked.

## 4. PROPERTIES OF THE ALGORITHM

This section shows some properties of the proposed algorithm.

**Lemma 1.** *If  $j$  is a successor of  $i$  in the WFG, then  $(j, \_) \in Succ_i$  upon termination of the algorithm.*

*Proof.* By induction on  $d(i, j)$ , the distance between node  $i$  and node  $j$  in the WFG.

Base case:  $d(i, j) = 1$ . In this case,  $j$  is a neighbor of  $i$  in the WFG, and  $(j, i) \in Succ_i$  by step (1.1). Inductive hypothesis:  $\forall i$ , if  $d(i, j) = n$  then  $(j, \_) \in Succ_i$ .

Inductive step: If  $d(i, j) = n + 1$  then there is a path from node  $i$  to node  $j$  in the WFG. Let  $k$  be

For node  $i$ :

- (3.1) **if**  $deadlock = false \wedge Paths \neq \emptyset \wedge \forall j, (i, j, \_) \in Paths \rightarrow (j, i, \_) \in Paths$   
**then**  $deadlock := true$   
(3.2) **if**  $deadlock = false \wedge \forall j \in Neighbors, deadlock_j = true$   
**then**  $deadlock := true$

Figure 5: The proposed algorithm, third phase

the neighbor of  $i$  in that path. Then,  $d(k, j) = n$  and, because of the inductive hypothesis,  $(j, \_) \in Succ_k$ . Since  $k$  is a neighbor of  $i$ , then  $(j, k) \in Succ_i$  by step (1.4).  $\square$

**Lemma 2.** *If  $(j, \_) \in Succ_i$  upon termination of the algorithm, then there is a path from  $i$  to  $j$  in the WFG.*

*Proof.* If there is a tuple  $(j, i)$  in  $Succ_i$ , then it was added by step (1.1) and  $j$  is a neighbor of  $i$  in the WFG.

If there is a tuple  $(j, n)$  in  $Succ_i$ , with  $n \neq i$ , then it was added by step (1.4) and there is a tuple  $(j, \_)$  in  $Succ_n$ . Since  $n$  is a neighbor of  $i$ , there is a path from  $i$  to  $n$  in the WFG. Inductively for  $n$ , it can be found a path from  $n$  to  $j$ . If there is no such path, then  $(j, \_)$  would be deleted from  $Succ_i$  by step (1.5). Thus, there is a path from  $i$  to  $j$  in the WFG.  $\square$

**Lemma 3.** *Upon termination of the algorithm, each node knows all paths of the WFG that start on itself.*

*Proof.* By Lemmas 1 and 2, node  $i$  knows exactly all its successors upon termination of the algorithm. By step (2.1), a path from  $i$  to every node in  $Succ_i$  is added to variable  $Paths$ . By step (2.2), paths to nodes that are no longer reachable from  $i$  are deleted.  $\square$

**Lemma 4.** *Upon termination of the algorithm, each node knows all paths that start on every successor.*

*Proof.* By Lemma 3, each node know all the paths that start on itself. Those paths are propagated to predecessor nodes in step (2.3), until every predecessor knows all the paths that start on every successor. If the WFG changes, by steps (2.2) and (2.4) the invalid paths are deleted, and changes are propagated.  $\square$

**Lemma 5.** *Upon termination of the algorithm, at least each node that causes deadlock knows that it is deadlocked.*

*Proof.* By Lemmas 3 and 4, node  $i$  knows all paths of the WFG that start on itself and on every successor. If the guard of step (3.1) is true, node  $i$  knows that it is part of a knot. By the definition of deadlock under the OR model, a knot is a necessary and sufficient condition for the existence of a deadlock. Thus, a node that knows that it is part of a knot knows that it causes deadlock.

By step (3.2), a node such that every one of its neighbors knows that is deadlocked, also updates its variable  $deadlock$  to  $true$ .

Thus, when the algorithm terminates, at least every process that causes deadlock has its variable  $deadlock$  set on  $true$ .  $\square$

It has been shown that the algorithm detects all deadlocks. Now, it will be shown that the algorithm does not detect false deadlocks.

**Lemma 6.** *The algorithm does not detect false deadlocks.*

*Proof.* If a process has its variable  $deadlock$  set on  $true$ , it has executed the step (3.1) or the step (3.2). For the guard of step (3.2) to be true at a node, another process must have executed the step (3.1). If there is not a knot in the WFG, the guard of step (3.1) would never be true at any node, by Lemmas 3 and 4. Then, if a process executes step (3.1), it is because there is a knot in the WFG.

No process can have its variable  $deadlock$  set on  $true$  if there is no knot in the WFG; thus, the algorithm does not detect false deadlocks.  $\square$

**Lemma 7.** *The algorithm terminates.*

*Proof.* If the node that runs the algorithm is not deadlocked, then it will eventually receive a resource for which it was waiting. Then, the guard at step (1.3) will be true, and the action will be executed. As the sets in variables  $Succ$  and  $Paths$  will be emptied, no other guard will be true at the node, and the algorithm terminates.

If the node that runs the algorithm is deadlocked, all its successors are deadlocked. Therefore, no new outgoing edges can be added or removed; the part of the WFG that the node can reach does not change. After a finite number of executions of steps (1.4) and (1.5), the guards of those steps

will not become true again. The same situation occurs with the guards of phases 2 and 3. Then, there will be no more moves to make, and the algorithm terminates.  $\square$

The next theorem follows from Lemmas 5, 6, and 7.

**Theorem 1.** *The proposed algorithm is complete and correct.*

## 5. DEADLOCK RESOLUTION

In order to resolve a deadlock, one of the deadlocked processes must be terminated. Therefore, once a deadlock is detected, it becomes necessary to choose a victim to terminate. Terminating just any process does not necessarily solve the deadlock. In Figure 2, if process 5 were terminated, the rest of the processes would still be deadlocked, because the knot in the WFG remains. It is necessary to terminate one process from each knot in the WFG.

Upon termination of the algorithm, each process that causes deadlock knows that it is deadlocked. The algorithm can be modified slightly, so that they can also know that are part of a knot. This effect can be accomplished by adding a boolean variable, say *knot*, initialized to *false*. If the guard of step (3.1) is true, then the variable is updated to *true*, because it is in this step that a process knows that it is part of a knot. Also, in step (1.3), the variable must be reset to *false*.

Once the algorithm is finished, the processes that are part of a knot can start an algorithm to choose a victim such that, when terminated, the deadlock is solved. For example, a leader-election algorithm as the one described by Ghosh and Gupta is enough [5].

After a process is terminated, variable *deadlock* should be reset to *false* at each node; also, all variables *knot* should be reset to *false*. The immediate predecessors of the terminated processes would see a change in their variable *Neighbors*, and the detection algorithm would recompute for the new WFG.

## 6. CONCLUDING REMARKS

This article presents a deadlock-detection algorithm for the OR requirement model. The algorithm is dynamic since it supports changes to the WFG while it is executing, such as addition and removal of nodes and edges. The algorithm is complete since it detects all deadlocks. The algorithm is correct since it detects no false deadlocks. Additionally, slight modifications to the algorithm that allow to resolve deadlocks once every causing node is identified have been discussed.

Since Dijkstra introduced the concept of self-stabilization in 1974 [3], several self-stabilizing algorithms to solve many problems in distributed systems have been proposed. Mutual exclusion and leader election are among the classical problems solved with this approach. Schneider has written a complete survey on the subject [12].

In general, a system is said to be self-stabilizing if, regardless of its initial global state, it can reach a legal global state in a finite number of steps [3]. The global state is the cartesian product of the local states of every processor in the system. The definition of legal and illegal global states depends on the context of the problem being solved.

The ability to regain a legal global state that those systems present, makes them able to support transient faults. A transient fault is one that occurs once and ceases to exist.

Some self-stabilizing algorithms to solve the deadlock-detection problem under the AND model have been proposed [4, 9]. Nevertheless, the classic definitions of some concepts used in self-stabilizing algorithms are not clearly applicable in the context of deadlock detection. The difficulty is in the definition of legal and illegal global states, and transient faults. Thus, the next step is to define those concepts suitably, in order to present an algorithm that supports transient faults, and that is self-stabilizing.

## ACKNOWLEDGEMENTS

Christian F. Orellana and María Pía Soto dedicate this work to the memory of their advisor and beloved friend Alvaro E. Campos, who unexpectedly passed away on July 25, 2003.

Alvaro was an excellent teacher, a remarkable researcher, and a wonderful person, who always worked with dedication, enthusiasm, and optimism. Meeting him was an enlightening experience and a genuine privilege. His many students will miss him much, and will continue with his legacy.

## 7. REFERENCES

- [1] Gabriel Bracha and Sam Toueg. A distributed algorithm for generalized deadlock detection. In *Symposium on Principles of Distributed Computing*, pages 285–301, Vancouver, British Columbia, Canada, August 1984.
- [2] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

- [3] Edsger Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
- [4] Mitchell Flatebo and Ajoy Kumar Datta. Self-stabilizing deadlock detection algorithms. In *Proceedings of the 1992 ACM Annual Conference on Communications*, pages 117–122, Kansas City, Missouri, April 1992.
- [5] Sukumar Ghosh and Arobinda Gupta. An exercise in fault-containment: self-stabilizing leader election. *Information Processing Letters*, 59(5):281–288, September 1996.
- [6] José R. González de Mendivil, Federico Fariña, José R. Garitagoitia, C. F. Alastruey, and J. M. Bernabeu-Auban. A distributed deadlock resolution algorithm for the AND model. *IEEE Transactions on Parallel and Distributed Systems*, 10(5):433–447, May 1999.
- [7] T. Herman and K. Chandy. A distributed procedure to detect AND/OR deadlock. Technical Report TR LCS-8301, Department of Computer Science, University of Texas, Austin, Texas, 1983.
- [8] Richard C. Holt. Some deadlock properties on computer systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.
- [9] Mehmet H. Karaata and Jeffery C. Line. Self-stabilizing algorithms for deadlock detection and identification in distributed systems. In *Proceedings of the ISCA Thirteenth International Conference on Parallel and Distributed Computing*, pages 320–325, Las Vegas, Nevada, August 2000.
- [10] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, December 1987.
- [11] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, SE-12(4):531–537, April 1986.
- [12] Marco Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
- [13] Loren Schwiebert. Deadlock-free oblivious wormhole routing with cyclic dependencies. *IEEE Transactions on Computers*, 50(9):865–876, September 2001.
- [14] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*. John Wiley & Sons, New York, NY, first edition, 2000.